Swan Chain Staking Audit Report V2

Signature: 8d7713732a27d7c485f2620356f15e9ffaeab6bfd2b105004fb91ecb6dd8d314

Swan Chain Staking Audit Report V2

Scope

Project Name	Swan Chain Staking
Repo	https://github.com/swanchain/swan-contracts/tree/release/v1.0.0-1/contracts/staking
Fixed	https://github.com/swanchain/swan-contracts/tree/release/v1.0.0-2/contracts/staking
Language	Solidity
Scope	contracts/staking/*.sol

Disclaimer

The audit does not ensure that it has identified every security issue in the smart contracts, and it should not be seen as a confirmation that there are no more vulnerabilities. The audit is not exhaustive, and we recommend further independent audits and setting up a public bug bounty program for enhanced security verification of the smart contracts. Additionally, this report should not be interpreted as personal financial advice or recommendations.

Auditing Process

- Static Analysis: We perform static analysis using our internal tools and Slither to identify potential vulnerabilities and coding issues.
- Fuzz Testing: We execute fuzz testing with our internal fuzzers to uncover potential bugs and logic flaws.
- Invariant Development: We convert the project into Foundry project and develop Foundry invariant tests for the project based on the code semantics and documentations.
- Invariant Testing: We run multiple fuzz testing tools, including Foundry and ItyFuzz, to identify violations of invariants we developed.
- Formal Verification: We develop individual tests for critical functions and leverage Halmos to prove the functions in question are not vulnerable.
- Manual Code Review: Our engineers manually review code to identify potential vulnerabilities not captured by previous methods.

Vulnerability Severity

We divide severity into three distinct levels: high, medium, low. This classification helps prioritize the issues identified during the audit based on their potential impact and urgency.

- **High Severity Issues** represent critical vulnerabilities or flaws that pose a significant risk to the system's security, functionality, or performance. These issues can lead to severe consequences such as fund loss, or major service disruptions if not addressed immediately. High severity issues typically require urgent attention and prompt remediation to mitigate potential damage and ensure the system's integrity and reliability.
- **Medium Severity Issues** are significant but not critical vulnerabilities or flaws that can impact the system's security, functionality, or performance. These issues might not pose an immediate threat but have the potential to cause considerable harm if left unaddressed over time. Addressing medium severity issues is important to maintain the overall health and efficiency of the system, though they do not require the same level of urgency as high severity issues.
- **Low Severity Issues** are minor vulnerabilities or flaws that have a limited impact on the system's security, functionality, or performance. These issues generally do not pose a significant risk and can be addressed in the regular maintenance cycle. While low severity issues are not critical, resolving them can help improve the system's overall quality and user experience by preventing the accumulation of minor problems over time.

Below is a summary of the vulnerabilities with their current status, highlighting the number of issues identified in each severity category and their resolution progress.

	Number	Resolved
High Severity Issues	2	2
Medium Severity Issues	1	1
Low Severity Issues	2	2
Info Severity Issues	1	1

Findings

[High] `removeStakingLimit` function lacks access control

The `removeStakingLimit` function lacks proper access control, allowing any address to call it and remove the staking limit. This could lead to unexpected behavior and potential manipulation of the staking system.

```
// contracts/staking/SwanStaking.sol
function removeStakingLimit() external {
    isStakingCapSet = false;
    stakableAmount = 0;
    emit StakingLimitRemoved();
}
```

Recommendations:

Implement proper access control for the `removeStakingLimit` function. Add `onlyAdmin` or `onlyOwner` modifiers to restrict access to authorized addresses only.

Status: Fixed

[High] Lack of token balance verification in the `initialization` function may cause arithmetic underflow

In the `SwanStaking` contract, the `_bootstrapInitialHolder()` function did not verify whether the contract actually held a sufficient number of tokens when setting the initial `bufferBalance`. This may cause arithmetic underflow when calculating the unbufferedBalance in the `withdraw()` function.

```
function _bootstrapInitialHolder(uint256 balance) internal {
    require(balance > 0);
    if (_getTotalShares() == 0) {
        bufferBalance = balance;       // @audit No verification of actual
token balance
        totalPooledSwan = balance;
        emit Stake(address(this), balance);
        _mintInitialShares(balance);
    }
}
```

in `withdraw()` function:

```
uint256 unbufferedBalance = IERC20(stakedToken).balanceOf(address(this)) -
bufferBalance;
```

If the actual token balance of the contract is less than `bufferBalance`, the subtraction operation will be rolled back.

Recommendations:

Add token balance verification in the `_bootstrapInitialHolder` function:

```
function _bootstrapInitialHolder(uint256 balance) internal {
    require(balance > 0);
    if (_getTotalShares() == 0) {
        // Verify that the contract actually holds enough tokens
        require(
            IERC20(stakedToken).balanceOf(address(this)) == balance,
            "Insufficient initial balance"
        );

        bufferBalance = balance;
        totalPooledSwan = balance;
        emit Stake(address(this), balance);
        _mintInitialShares(balance);
    }
}
```

Status: Fixed

[Med] After `postRewards` is called, users with a small `stake` amount may get 0 shares

At the beginning of the project, `_getTotalShares()` and `_getTotalPooledSwan()` are basically equal, and there will be no situation where users with a small `stake` amount may get 0 shares. However, there is a `postRewards` function in the contract. When the administrator calls `postRewards`, the value of `totalPooledSwan` will be increased. At this time, users with a small `stake` amount may get 0 shares.

The test code is as follows:

```
function testStakingAtfterPostRewards() public {
    stakingContract.addAdmin(owner);
    stakingContract.postRewards(10 ether);

    stakedToken.mint(address(user1), 10 ether);
    vm.prank(user1);
    stakedToken.approve(address(stakingContract), 10 ether);
    vm.prank(user1);
    stakingContract.stake(10);

    uint256 sharesAfterStake = stakingContract.balanceOf(user1);
    console2.log("User1 shares after staking 10:", sharesAfterStake);
}
```

```
Ran 1 test for test/test_Staking.t.sol:StakingTest
[PASS] testStakingAtfterPostRewards() (gas: 186966)
Logs:
  Setting up test
  User1 shares after staking 10: 0
```

Recommendation:

```
diff --git a/contracts/staking/SwanStaking.sol b/contracts/staking/SwanStakin
g.sol
index 8b251d6..10da6d8 100644
--- a/contracts/staking/SwanStaking.sol
+++ b/contracts/staking/SwanStaking.sol
@@ -206,6 +206,7 @@ contract Staking is Initializable, OwnableUpgradeable, UU
PSUpgradeable, Reentran
         IERC20(stakedToken).safeTransferFrom(msg.sender, address(this), _sta
keAmount);

         uint256 sharesAmount = getSharesByPooledSwan(_stakeAmount);
+        if(sharesAmount == 0){revert()};
         _mintShares(msg.sender, sharesAmount);

         bufferBalance += _stakeAmount;
```

Status: Fixed

[Low] Signature replay attack vulnerability in `claim` function

The `claim` function has a signature replay attack vulnerability. Although the function checks whether the signature has been used, nonce or other unique identifier is included in the signature message. This allows an attacker to replay a valid signature in a different chain or a different contract instance, potentially leading to unauthorized funds.

An attacker observes a valid `claim` transaction on one chain or contract instance. They can then replay the exact same transaction (including signatures) on another chain where the same contract is deployed, or in a new instance after the contract is redeployed. If the signer's address is still authorized as an administrator in the new environment, the replay attack will succeed, allowing the attacker to claim funds they do not deserve.

```
// contracts/staking/NativeClaim.sol
function claim(string memory timstampStr, uint amount, bytes memory
signature)
    external
    whenNotPaused // Add this modifier
    validateSignatureWithAmount(timstampStr, msg.sender, amount, signature)
{
    require(!isSignatureUsed[signature], "signature already used");
    isSignatureUsed[signature] = true;
    (bool sent, bytes memory data) = msg.sender.call{value: amount}("");
    require(sent, "Failed to send Ether");

    emit Claimed(timstampStr, msg.sender, amount);
}
```

Recommendations:

To prevent this vulnerability, the chain ID and contract address should be included in the signature message. Also consider adding a nonce that increments with each claim for each user.

Status: Fixed

[Low] Share-based token allowance check may fail due to rebasing mechanism

In the `fSwan` contract, the `transferSharesFrom()` function checks token allowances based on the current token amount converted from shares. However, since the token implements a rebasing mechanism where the token/share ratio can change, the actual token amount at transfer time may exceed the previously approved amount, causing the transaction to fail.

```
function transferSharesFrom(
    address _sender,
    address _recipient,
    uint256 _sharesAmount
) external returns (uint256) {
    uint256 tokensAmount = getPooledSwanByShares(_sharesAmount); // @audit
token amount may increase
    _spendAllowance(_sender, msg.sender, tokensAmount);
    _transferShares(_sender, _recipient, _sharesAmount);
    _emitTransferEvents(_sender, _recipient, tokensAmount, _sharesAmount);
    return tokensAmount;
}
```

Impact

- If the total staked SWAN increases between approval and transfer, the same number of shares will represent more tokens

Recommendations:

Implement share-based allowances instead of token-based.

Status: Acknowledged

[Info] To prevent silent failure, use `safeTransfer` instead of `transfer`

The `SwanStaking::deposit` function uses `transfer` instead of `safeTransfer` when transferring stakedToken. This may cause the transfer to fail silently due to the lack of return value check. It is recommended to use `safeTransfer` instead of `transfer`.

```
//contracts/staking/SwanStaking.sol
    function deposit() external onlyAdmin {
        uint256 depositsValue = bufferBalance;
        bufferBalance = 0;

        IERC20(stakedToken).transfer(poolWallet, depositsValue);
        emit Unbuffered(depositsValue);

    }
```

Recommendation:
use `safeTransfer` instead of `transfer`

Status: Fixed